

SETUP & CREATING ARRAYS	
<code>import numpy as np</code>	standard import (NumPy 2.x)
<code>np.array([1, 2, 3])</code>	from a list (nested list = 2D)
<code>np.zeros((2, 3))</code>	2×3 of 0.0
<code>np.ones((2, 3))</code>	2×3 of 1.0
<code>np.full((2, 2), 7)</code>	filled with a constant
<code>np.arange(0, 10, 2)</code>	0,2,4,6,8 (stop excluded)
<code>np.linspace(0, 1, 5)</code>	5 evenly spaced pts, stop included
<code>np.eye(3)</code>	3×3 identity matrix
<code>np.zeros_like(a)</code>	same shape/dtype as a

RANDOM NUMBERS (GENERATOR)	
<pre>rng = np.random.default_rng(42) rng.random((2, 3)) # uniform [0, 1) rng.integers(0, 10, 5) # ints, high excluded rng.normal(0, 1, 100) # mean, std, size rng.choice(a, 3, replace=False) rng.shuffle(a) # in place rng.permutation(10) # shuffled copy</pre>	
Prefer <code>default_rng(seed)</code> over legacy <code>np.random.seed</code> .	

ATTRIBUTES & TYPES	
<code>a.shape</code>	dimensions, e.g. (3, 4)
<code>a.ndim</code>	number of axes
<code>a.size</code>	total element count
<code>a.dtype</code>	element type (int64, float64...)
<code>a.astype(np.float32)</code>	converted copy
<code>np.array([1, 2], dtype=float)</code>	set dtype on creation

INDEXING & SLICING	
<pre>a = np.arange(10) # 1D a[0] a[-1] a[2:5] a[::2] a[::-1] m = np.arange(12).reshape(3, 4) # 2D m[1, 2] # row 1, col 2 m[0] # first row m[:, 1] # column 1 m[1:, :2] # rows 1+, cols 0-1</pre>	

MASKS & FANCY INDEXING	
<pre>a[a > 5] # boolean mask a[(a > 2) & (a < 8)] # & ~ , need () a[[0, 3, 5]] # fancy: pick positions m[[0, 2], [1, 3]] # pairs (0,1),(2,3) a[a > 5] = 0 # assign via mask (a > 5).sum() # count True</pre>	
Slices are views ; masks and fancy indexing return copies .	

RESHAPE & TRANSPOSE	
<code>a.reshape(3, 4)</code>	new shape; -1 infers a size
<code>a.reshape(-1, 1)</code>	column vector (n, 1)
<code>a.ravel()</code>	flatten (view if possible)
<code>a.flatten()</code>	flatten (always a copy)
<code>m.T / np.transpose(m)</code>	swap axes
<code>a[:, np.newaxis]</code>	add axis: (n,) to (n, 1)
<code>np.squeeze(a)</code>	drop length-1 axes

STACK & SPLIT	
<code>np.concatenate([a, b], axis=0)</code>	join along existing axis
<code>np.vstack([a, b])</code>	stack rows (vertical)
<code>np.hstack([a, b])</code>	stack columns (horizontal)
<code>np.column_stack([x, y])</code>	1D arrays become columns
<code>np.split(a, 3)</code>	3 equal parts (list of arrays)
<code>np.split(a, [2, 5])</code>	split at indices 2 and 5

MATHS & UFUNCS	
<code>a + b, a - b, a * b, a / b</code>	element-wise
<code>a ** 2, a % 3, a // 3</code>	power, modulo, floor div
<code>np.sqrt(a), np.exp(a)</code>	root, e ^x
<code>np.log(a), np.log10(a)</code>	natural / base-10 log
<code>np.abs(a), np.round(a, 2)</code>	absolute, rounding
<code>np.sin(a), np.cos(a)</code>	trig (radians)
<code>np.maximum(a, b)</code>	element-wise max of two
<code>np.cumsum(a), np.diff(a)</code>	running total, differences

AGGREGATIONS & AXIS	
<code>a.sum()</code> <code>a.mean()</code> <code>a.std()</code>	over all elements
<code>a.min()</code> <code>a.max()</code> <code>a.var()</code>	also <code>np.median(a)</code> , <code>np.percentile(a, 90)</code>
<code>m.sum(axis=0)</code>	collapse rows: one value per column
<code>m.sum(axis=1)</code>	collapse columns: one value per row
<code>m.mean(axis=1, keepdims=True)</code>	keep 2D shape for broadcasting
<pre>m = [[1, 2, 3], [4, 5, 6]] m.sum(axis=0) -> [5 7 9] m.sum(axis=1) -> [6 15]</pre>	

BROADCASTING	
Align shapes from the right. Dims are compatible if equal or one is 1; size-1 dims are stretched.	
<pre>(3, 4) + (4,) -> (3, 4) OK (3, 4) + (3,) -> error (3, 4) + (3, 1) -> (3, 4) OK (3, 1) + (1, 4) -> (3, 4) OK # centre columns X - X.mean(axis=0) # row-wise scale X / X.sum(axis=1, keepdims=True)</pre>	

WHERE / SELECT / CLIP	
<code>np.where(a > 0, a, 0)</code>	if-else per element
<code>np.where(a > 5)</code>	indices where True (tuple)
<code>np.select([c1, c2], [v1, v2], default=0)</code>	multi-way conditions
<code>np.clip(a, 0, 10)</code>	limit values to [0, 10]
<code>np.any(a > 5), np.all(a > 0)</code>	test whole array

UNIQUE, SORT & SEARCH	
<code>np.unique(a)</code>	sorted unique values
<code>v, n = np.unique(a, return_counts=True)</code>	values and their counts
<code>np.sort(a) / a.sort()</code>	sorted copy / in place
<code>np.argsort(a)</code>	indices that would sort a
<code>a[np.argsort(a)[::-1]]</code>	descending order
<code>a.argmax()</code> <code>a.argmin()</code>	index of max / min
<code>m.argmax(axis=1)</code>	best column per row

LINEAR ALGEBRA	
<code>A @ B</code>	matrix multiply (preferred)
<code>np.dot(a, b)</code>	dot product / matmul
<code>np.linalg.inv(A)</code>	inverse (square matrix)
<code>np.linalg.solve(A, b)</code>	solve Ax = b (better than inv)
<code>np.linalg.det(A)</code>	determinant
<code>np.linalg.norm(v)</code>	Euclidean length
<code>np.linalg.eig(A)</code>	eigenvalues, eigenvectors

MISSING VALUES (NAN)	
<code>np.nan</code>	float "not a number"
<code>np.isnan(a)</code>	boolean mask of NaNs
<code>a[~np.isnan(a)]</code>	drop NaNs
<code>np.nanmean(a), np.nansum(a)</code>	ignore NaNs (also <code>nanstd</code> , <code>nanmax</code>)
<code>np.nan_to_num(a, nan=0)</code>	replace NaN
<code>a.mean()</code>	returns nan if any NaN is present.

COPIES VS VIEWS	
<pre>b = a[1:4] # view: shares memory b[0] = 99 # also changes a! c = a[1:4].copy() # independent np.shares_memory(a, b) # True</pre>	
Views: slices, reshape, ravel, T. Copies: masks, fancy indexing, flatten, astype.	

HANDY EXTRAS	
<code>np.corrcoef(x, y)</code>	correlation matrix
<code>np.isin(a, [1, 2])</code>	membership mask
<code>np.meshgrid(x, y)</code>	coordinate grids
<code>np.save("a.npy", a); np.load("a.npy")</code>	binary file I/O
<code>np.loadtxt("f.csv", delimiter=",")</code>	read numeric text

TIPS	
Vectorise, do not loop. <code>a * 2</code> is ~50x faster than a Python for loop.	
Use <code>np.where</code> / masks instead of if-statements in loops.	
Broadcast instead of <code>np.tile</code> ; use <code>keepdims=True</code> .	
Use float32 to halve memory on big arrays.	
Check <code>a.shape</code> and <code>a.dtype</code> first when results look odd.	